

ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs

Carlos Luque¹, Miquel Moreto^{1,2}, Francisco J. Cazorla^{1,3},
Roberto Gioiosa¹ and Mateo Valero^{1,2}

¹Barcelona Supercomputing Center (BSC), Barcelona, Spain. {carlos.luque, roberto.gioiosa}@bsc.es

²Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. {mmoreto, mateo}@ac.upc.edu

³Spanish National Research Council (IIA-CSIC), Spain. {francisco.cazorla}@bsc.es

Resumen

Chip-MultiProcessors (CMP) introduce complexities when accounting CPU utilization to processes because the progress done by a process during an interval of time highly depends on the activity of the other processes it is co-scheduled with. We propose a new hardware CPU accounting mechanism to improve the accuracy when measuring the CPU utilization in CMPs and compare it with previous accounting mechanisms. Our results show that currently known mechanisms lead to a 16% average error when it comes to CPU utilization accounting. Our proposal reduces this error to less than 3% in a modeled 8-core processor system.

1. Introduction

The Operating System (OS) provides the user with an abstraction of the hardware resources. The user application perceives this abstraction as if it is using the complete hardware while, in fact, the OS shares hardware resources among the user applications. Hardware resources can be shared *temporally* and *spatially*. Hardware resources are time shared between users when each task can make use of a resource for a limited amount of time (for example, the exclusive use of a CPU). Orthogonally, hardware resources can be shared spatially when each task makes use of a limited amount of resources, such as the cache memory or the I/O bandwidth.

The execution time of an application is in-

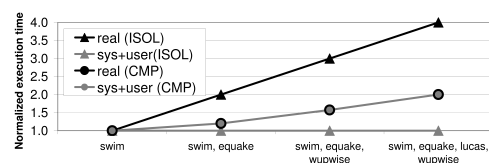


Figure 1: Total (*real*) and accounted (*sys+user*) time of *swim* in different workloads running on an Intel Xeon Quad-Core CPU

fluenced by the amount of hardware resources shared with the other running applications. It is also affected by how long the application runs with other applications. However, *the time accounted to that application is always the same regardless of the workload¹ in which it is executed, i.e., regardless of how many applications are sharing the hardware resources at any given time*. We call this principle, the *Principle of Accounting*. Unix-like systems differentiate the real execution time and the time an application actually is running on a CPU. Commands such as `time` or `top` provide three values: *real*, *user* and *sys*. *Real* is the elapsed wall clock time between invocation and termination of the application; *user* is the time spent by the application in the *user mode*; and *sys* is the time spent in the *kernel mode* on behalf of the application. In these systems, *sys+user* time is the execution time accounted to the application.

¹A workload is a set of applications running, simultaneously, on a CPU

Figure 1 shows the total (*real*) and the accounted execution time (*sys+user*) of the 171.*swim* (or simply *swim*) SPEC CPU 2000 benchmark [1] when it runs in different workloads. In this figure, the time results are normalized to the real execution time of *swim* when it runs in isolation (ISOL). For this experiment, we use an Intel Xeon Quad-Core processor at 2.5 GHz (though the general trends drawn from Figure 1 apply to all current CMPs), which has four cores in the chip on which we run Linux 2.6.18. We move all the OS activity to the first core, leaving the other cores as isolated as possible from OS noise. When *swim* runs alone in one of the isolated cores, it completes its execution in 117 seconds. However, when *swim* runs together with other applications in the same core, its real execution time increases up to 4x due to context switches done by the OS (black triangles in Figure 1). Nevertheless, *swim* is accounted roughly the same time (grey triangles), which is the time the application actually uses the CPU. Applications may suffer some delay because they lose part of the cache and TLB contents on every context switch, but this effect is small in this case. Hence, even if *swim*'s total execution time increases depending on the other applications it is co-scheduled with, the time accounted to *swim* is always the same.

However, processors with shared on-chip resources, such as CMPs [2], make CPU accounting more complex because the progress of an application depends on the activity of the other applications running at the same time. Current OSs still use the CA for multicore processors, which can lead to inaccuracy for the time accounted to each application. In order to show this inaccuracy, in a second experiment, we use all the cores in the Intel Xeon Quad-Core processor. Next, we execute *swim* with several workloads as shown by the x-axis in Figure 1. In this case, *swim* suffers no time sharing and *real* time is roughly the same as *sys+user* because the number of tasks that are running is equal or less than the number of virtual CPUs (cores) in the system. In Figure 1, the grey circles show a variance up of to 2x in the time accounted to *swim* depending

on the workload in which it runs. This means that (at least with current known open source OSs such as Linux) *an application running on a CMP processor may be accounted different CPU utilization according to the other applications running on the same chip at the same time*. From the user point of view this is an undesirable situation, as the same application with the same data input set is accounted differently depending on the applications it is co-scheduled with.

CPU accounting affects several key components of a computing system: First, if the OS scheduler does not properly account the CPU utilization of each application, the OS scheduling algorithm will fail to maintain fairness between applications. As a consequence, the scheduling algorithm cannot guarantee that an application progresses with its work as expected. Second, CPU accounting can be also used in data centers to charge users, together with other factor such as used amount of memory, disk space, I/O activity, etc.

The main contributions of this paper are: For the first time, we provide a comprehensive analysis of the CPU accounting accuracy of the CA. To the best of our knowledge, the CA is the only accounting mechanism for CMPs for open source OSs such as Linux. Next, we propose a hardware mechanism, *Inter-Task Conflict-Aware* (ITCA) accounting [3], which improves the accuracy of the CA for CMPs. When running on a modeled 2-, 4-, and 8-core CMP, ITCA reduces the off estimation of the CA from 7.0 %, 13 %, and 16 %, to 2.4 %, 3.7 %, and 2.8 %, respectively.

The rest of the paper is organized as follows. Section 2 analyses and formalizes the CPU accounting problem. Section 3 describes our proposed CPU accounting mechanism with improved accuracy. The experimental methodology and the results of our simulations are presented in Section 4. Section 5 discusses related work and, finally, Section 6 concludes the paper.

2. Formalizing the problem

Currently, the OS perceives different cores in a CMP as multiple independent virtual CPUs. The OS does not consider the interaction between tasks caused by shared hardware resources in the CA. However, the time running on a virtual CPU is not an accurate measure of the amount of CPU resources the task has received. The CPU time to account to a task in a CMP processor does not only depend on the time that task is scheduled onto a CPU, but also on the progress it makes during that time. In our view, CMP processors, have to maintain the same *principle of accounting* that rules today in uniprocessor and Symmetric MultiProcessors (SMP) systems² accounting: the CPU accounting of a task should be independent from the workload in which this task runs. For example, let's assume that a task X runs for a period of time in a CMP (TR_{X,I_X}^{CMP}), in which it executes I_X instructions. It is our position that the actual time to account this task, denoted TA_{X,I_X}^{CMP} , should be the time it would take this task to execute these I_X instructions in isolation, denoted TR_{X,I_X}^{ISOL} .

The relative progress that task X has in this interval of time (P_{X,I_X}^{CMP}) can be expressed as $P_{X,I_X}^{CMP} = TR_{X,I_X}^{ISOL} / TR_{X,I_X}^{CMP}$. The relative progress can also be expressed as $P_{X,I_X}^{CMP} = IPC_{X,I_X}^{CMP} / IPC_{X,I_X}^{ISOL}$ in which IPC_{X,I_X}^{CMP} and IPC_{X,I_X}^{ISOL} are the IPC of task X when executing the same I_X instructions in the CMP and in isolation, respectively. Then, $TA_{X,I_X}^{CMP} = TR_{X,I_X}^{CMP} \cdot P_{X,I_X}^{CMP}$, from which we conclude $TA_{X,I_X}^{CMP} = TR_{X,I_X}^{ISOL}$. This follows our principle of workload-independent accounting.

When using this approach to measure CPU accounting, the main issue to address is how to determine dynamically (while a task X is simultaneously running with other tasks) on each context switch, the time (or IPC) it will take X to execute the same instructions if it is alone in the system. An intuitive solution to this problem is to provide hardware mechanisms to determine the IPC in isolation of each task running in a workload by periodical-

ly running each task in isolation [4][5]. By averaging the IPC in the different isolation phases, an accurate measurement of the IPC in isolation of the task can be obtained. However, as the number of tasks simultaneously executing in a multicore processor increases to dozens or even hundreds, this solution will not scale, as the number of isolation phases increases linearly with the number of tasks in the workload. As a consequence, the time the task runs in CMP architectures is reduced, affecting the system performance.

Throughout this paper, we refer to *inter-task* resource conflicts to those resource conflicts that a task suffers due to the interference of the other tasks running at the same time. For example, a given task X suffers an inter-task L2 cache miss when it accesses a line that was evicted by another task, but would have been in cache, if X had run in isolation. Likewise, *intra-task* resource conflicts denote those resource conflicts that a task suffers even if it runs in isolation. These are conflicts inherent to the task.

The CA accounts tasks based on the time they run on a CPU, instead of the progress each task does. Therefore, the CA implicitly assumes that running tasks have full access to the processor resources. However, each task shares resources with other tasks when running in a CMP, which leads to inter-task conflicts. As a consequence, a task takes longer to finish its execution than when it runs in isolation, resulting in longer accounting time. For this reason for a task X, the CA leads to *over-estimation* $TA_{X,I_X}^{CA} = TR_{X,I_X}^{CMP} \geq TR_{X,I_X}^{ISOL}$. A task has no over-estimation only if it executes with no slowdown in CMP with respect to its execution in isolation, in which case $TA_{X,I_X}^{CA} = TR_{X,I_X}^{CMP} = TR_{X,I_X}^{ISOL}$.

The main source of over-estimation in our CMP baseline architecture are inter-task conflicts and, in particular, inter-task L2 misses.

3. Inter-Task Conflict-Aware Accounting

The target of our proposal is to accurately estimate the time accounted to a task in CMPs.

²SMPs are systems with several single thread, single core chips

The basic idea of ITCA is to account to a task only those cycles in which the task is not stalled due to an inter-task L2 cache miss. In other words, a task is accounted CPU cycles when it is progressing or when it is stalled due to an intra-task L2 miss. The next paragraphs provide a detailed discussion of when the accounting of a task is stopped and resumed.

L2 data misses: We consider a task is in one of the following states: (s1) It has no L2 (data) cache misses or it has only intra-task L2 misses in flight; (s2) It has only inter-task L2 misses in flight; and (s3) It has both inter-task and intra-task L2 misses in flight simultaneously.

In the state (s1) we do a normal accounting because there is not either an inter-task L2 miss or an intra-task L2 miss. We consider a task is not progressing, and hence, it should not be accounted in state (s2). In other words, accounting is stopped when the task experiences an inter-task L2 miss and it cannot overlap its stall with any other intra-task L2 miss. We resume accounting the task when the inter-task L2 miss is resolved or the task experiences an intra-task L2 miss, in which case the task is able to overlap the memory latency of the inter-task L2 miss with at least one intra-task L2 miss.

In the state (s3), we have that the inter-task L2 miss overlaps with another intra-task L2 miss. As a consequence, in general we do a normal accounting to the task in that state. However, when an inter-task L2 miss becomes the oldest instruction in the Reorder Buffer (ROB) and the register renaming is stalled, the task loses an opportunity to extract more Memory Level Parallelism (MLP). For example, let's assume that there are S instructions between the inter-task L2 miss in the top of the ROB and the next intra-task L2 miss in the ROB. In this situation, if the task had not experienced the inter-task L2 miss it would have executed the S instructions after the last instruction currently in the ROB. Any L2 miss in those S instructions would have been sent to memory, increasing the MLP. We take care of this lost opportunity of extracting MLP by stopping the accounting of a task if the instruction in the top of the ROB is an inter-

task L2 miss and the ROB is full. We call this condition state (s4).

L2 instruction misses: Another condition in which we stop the accounting of a task, is when the ROB is empty because of an inter-task L2 cache instruction miss (s5). In our processor setup instruction cache misses do not overlap with other instruction cache misses. That is, at every instant, we have only 1 in flight instruction miss per task at most. Hence, on an inter-task instruction L2 miss we consider that the task is not progressing because of an inter-task conflict, and hence, we stop its accounting.

3.1. Implementation

Figure 2 shows a sketch of the hardware implementation of our proposal, which makes use of several *hardware resource status indicators*. Next, we explain in depth the different parts of our approach.

Detecting inter-task misses: We keep an *Auxiliary Tag Directory* (ATD) [6] for each core (see Figure 2 (a)). The ATD has the same associativity and size as the tag directory of the shared L2 cache and uses the same replacement policy. It stores the behavior of memory accesses per task in isolation. While the tag directory of the L2 cache is accessed by all tasks, the ATD of a given task is only accessed by the memory operations of that particular task. If the task misses in the L2 cache and hits in its ATD, we know that this memory access would have hit in cache if the task had run in isolation [7]. Thus, it is identified as an inter-task L2 miss.

Tracking inter-task misses: We add one bit called $ITdata_i$ in each entry i of the Miss Status Hold Register (MSHR). The $ITdata$ bit is set to one when we detect an inter-task data miss. Each entry of the MSHR keeps track of an in flight memory access from the moment it misses in the data L1 cache until it is resolved.

On a data L1 cache miss, we access the L2 tag directory and the ATD of the task in parallel. If we have a hit in the ATD and a miss in the L2 tag directory, we know that this is an inter-task L2 cache miss. Then, the $ITdata$ bit of the corresponding entry in the MSHR is

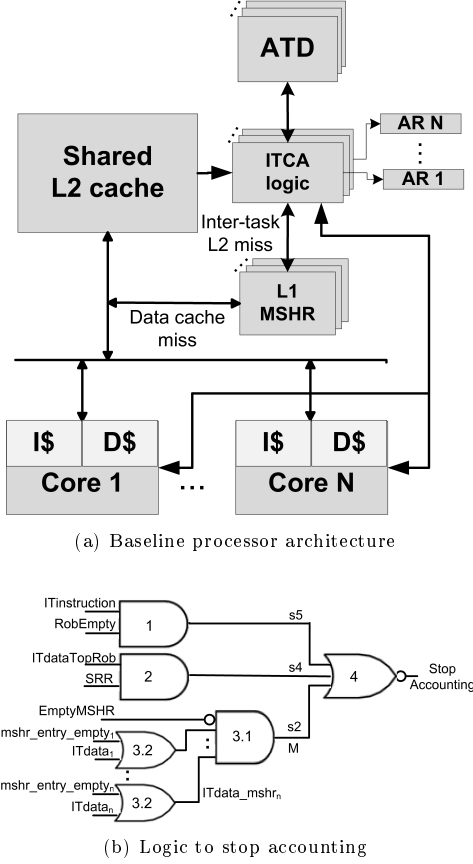


Figure 2: Hardware required for ITCA

set to 1. Once the memory access is resolved, we free its entry in the MSHR.

When the ROB is empty due to an inter-task L2 cache instruction miss, we stop accounting cycles to this task. For our purpose, we use a bit called *ITInstruction* that indicates whether the task has an inter-task L2 cache instruction miss or not.

Accounting CPU time: We stop the accounting of a given task when: First, the ROB is empty because of an L2 cache instruction miss (gate (1) in Figure 2 (b) that implements condition (s5)). *RobEmpty* is a signal that is already present in most processor architectures, while *ITInstruction* indicates whether or not a task has an L2 cache instruction miss.

Second, The oldest instruction in the ROB is an inter-task L2 cache data miss and we have a Stall in the Register Renaming (SRR), in which case *SRR* equals 1 (gate (2) in Figure 2 (b) that implements condition (s4)). Storing a bit to track inter-task L2 misses might require one bit per ROB entry. Third, all the occupied MSHR entries belong to inter-task misses. To determine this condition, we check whether every entry i of the MSHR is not empty ($mshr_entry_empty_i = 0$) and contains an inter-task L2 miss ($ITdata_i = 1$) (gates (3.1) and (3.2) in Figure 2 (b) implement condition (s2)). By making an AND operation of $ITdata_mshr_i$ and a signal showing whether the entire MSHR is empty, *EmptyMSHR* (3.1), we determine if we have to stop the accounting for the task. Finally, if any of the gates (1), (2) or (3.1) returns 1, we stop the accounting. Otherwise, we account the cycle normally to the task as occurs in states (s1) and (s3).

In a 2-core CMP, ITCA accounts for every spent cycle in three possible ways: (1) Each task is accounted for the cycle when both tasks progress (the cycle is accounted twice, one for each task). (2) Only one task is progressing and the cycle is accounted only to it. (3) The cycle is not accounted to any task when none of them is progressing.

The cycles accounted to each task in each core are saved into a special purpose register per core, *Accounting Register* or *AR* (see Figure 2 (a)), which can be communicated to the OS. This register is a read only register like the *Time Stamp Register* in Intel architectures. From the OS point of view working with ITCA is similar to working with the CA. On every context switch, the OS reads the Accounting Register of each task i (AR_i), where AR_i reports the time to account this task. With this information, the OS updates metrics of the system and carries out the scheduling tasks.

4. Experimental results

4.1. Experimental environment

We use MPsim [8], a trace driven CMP simulator to model three processor setups: a 2-core

CMP with a 2MB L2 cache, a 4-core CMP with a 4MB L2 cache and an 8-core CMP with an 8MB L2 cache. Each core is single threaded, has an 11-stage-deep pipeline and can fetch up to 8 instructions each cycle. Each core has 6 integer (I), 3 floating point (FP), and 4 load/store functional units; 64-entry I, FP, and load/store instruction queues; 512-entry reorder buffer and 196 I/FP physical registers. We use a two-level cache hierarchy with 128B lines with a separate 64KB, 2-way instruction cache and a 32KB, 4-way data cache and a 16-way L2 cache that is shared among all cores. The latency from L1 to L2 is 12 cycles, and from L2 to memory 300 cycles.

We feed our simulator with traces collected from the whole SPEC CPU 2000 benchmark suite using the reference input set. Each trace contains 300 million instructions, selected using SimPoint [9]. From these benchmarks, we generate 2-task, 4-task and 8-task workloads. In each workload, the first thread in the tuple is the Principal Thread (PTh) and the remaining threads are considered Secondary Threads (SThs). In every workload, we execute the PTh until completion. The other threads are re-executed until PTh completes. We characterize the results of our proposal based on the type of the PTh and SThs. We distinguish three combinations of ST: ILP, MEM and MIX. ILP combination contains only ILP benchmarks, MEM combination contains only memory-bound benchmarks and MIX combination contains a mixture of both.

As the main metric, we measure how off is the estimation provided by each accounting mechanism. The *off estimation* (relative error of the approximation) compares the accounted time of a particular accounting approach for the PT with the actual time it should be accounted for. The ratio $|1 - (TA_{PT, I_{PT}}^{CMP} / TR_{PT, I_{PT}}^{ISOL})|$ estimates the off estimation for a given accounting mechanism. The ratio $|1 - (TR_{PT, I_{PT}}^{CMP} / TR_{PT, I_{PT}}^{ISOL})|$ provides the off estimation for the CA. For each accounting policy, we also report the average values of the five workloads with the worst off estimation, denoted *Avg5WOE*.

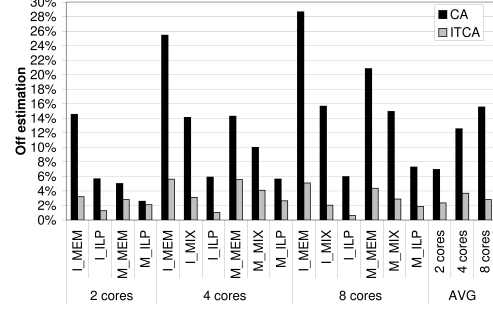


Figure 3: Off estimation of the CA and ITCA for 2-, 4- and 8-core CMPs with a shared 2MB, 4MB and 8MB L2 cache, respectively

4.2. Accuracy results

Figure 3 shows the off estimation of ITCA and the CA for our 3 processor setups. In this case, we show the average results of each group as we described in Section 4.1. The bars labeled AVG represent the average of each CMP configuration for all the groups. While on average the CA has an off estimation of 7.0 % (2 cores), 13 % (4 cores) and 16 % (8 cores), ITCA reduces it to less than 2.4 % (2 cores), 3.7 % (4 cores) and 2.8 % (8 cores). These results indicate that ITCA provides a good measure of the progress each task makes with respect to its execution in isolation, since ITCA takes into account inter-task L2 misses. Moreover, ITCA reduces the inaccuracy in the worst five cases: the Avg5WOE metric is 117 % (2 cores), 91 % (4 cores) and 94 % (8 cores) for the CA and only 32 % (2 cores), 35 % (4 cores) and 20 % (8 cores) for ITCA.

Next, we observe that the accuracy of the CA is worse when the PT is in the ILP group and any of the STs is in the MEM group. This is due to the fact that some of the ILP tasks experience a lot of hits in the L2 cache when they run in isolation, and when they run with MEM tasks, which make an intensive use of the L2 cache, the ILP tasks suffer a lot of inter-task misses. As a consequence, the ILP task suffers an increase in its execution time, which affects the accuracy of the CA. When the PT is in the MEM group, it already suffers a lot

of L2 misses in isolation, so that the increase in the number of L2 misses when it runs with other MEM tasks is relatively lower.

Next, we observe that the inaccuracy of the CA for a given group increases with the number of cores. Even if in our processor setups for 2, 4, and 8 cores the average cache space per task is kept the same (1MB per task), the average off estimation of the CA increases from 7.0% (2 cores) to 16% (8 cores). The main reason for that behavior is that having more tasks sharing the cache increases the probability that one of them thrashes the other tasks, which will lead to higher off estimations in the CA. The capacity of the L2 cache is not enough to store all the data of the tasks running simultaneously and for example, the off estimation of the group I_MEM in 2 cores is 14% but 29% in 8 cores.

4.3. Hardware proposals to provide fairness

Several hardware approaches deal with the problem of providing fairness in multicore architectures. Although, fairness is a desirable characteristic of a system, next we show that it cannot be used to provide an accurate CPU accounting. There are two main flavors of fairness. First, it is assumed that an architecture is fair when it gives the *same amount of resources* to each running task. However, ensuring a fixed amount of resources to a task [10, 5], does not translate into a CPU utilization that can be computed for that task. This is due to fact that the relation between the amount of resources assigned to a task and its performance can be different for each task. The second flavor of fairness considers that an architecture is fair when all tasks running on that architecture make *the same progress*. For example, let's assume a 2-core CMP with tasks X and Y. The system is said to be fair if in a given period of time, the progress made by X and Y is the same, $P_X = P_Y$. However, the fact that $P_X = P_Y$ does not provide a quantitative value that can be provided to the OS, so that it can account CPU time to each task. In other words, to know that $P_X = P_Y$ does not provide any information about CPU accounting

since P_X can be any value lower than 1. Therefore, systems providing fairness require an accurate CPU accounting mechanism as well.

5. Related work

We are not aware of any other work which studies CPU accounting for CMP architectures. Thus, ITCA is the first accounting mechanism designed specifically for CMP processors. For SMT processors [11], other proposals have been made. The IBM POWER5TM processor (a dual-core and 2-context SMT processor) includes a per-task accounting mechanism called *Processor Utilization of Resources Register* (PURR) [12]. The PURR approach estimates the time of a task based on the number of cycles the task can decode instructions: each POWER5 core can decode instructions from up to one task each cycle. The PURR accounts a given cycle to the task that decodes instructions that cycle. If no task decodes instructions on a given cycle, both tasks running on the same core are accounted one half of cycle. An improvement of PURR, denoted *scaled PURR* (SPURR) [13], is implemented in the IBM POWER6TM chip, which uses pipeline throttling and DVFS. SPURR provides a scaled count that compensates the impact of throttling and DVFS. ITCA can work in environments in which cores work at different frequencies with no change in its philosophy. The only effect seen by ITCA is a difference in the memory latency. Throttled cycles are simply not accounted to any task.

As a part of our future work we plan to explore CMP architectures in which each core is SMT. In this type of architectures we need to combine some of the solutions mentioned above for SMT architectures with our ITCA proposal for CMP processors.

6. Conclusions

CMP architectures introduce complexities in the CPU accounting because the progress done by a task varies depending on the activity of the other tasks running at the same time. The

current accounting mechanism, the CA, introduces inaccuracies when applied in CMP processors. This accounting inaccuracy may affect several key elements of the system such as the OS task scheduling or the charging mechanism in data centers. In this paper, we present a hardware support for a new accounting mechanism called *Inter-Task Conflict-Aware* (ITCA) accounting that improves the accuracy of the CA. In a 2-, 4- and 8-core CMP architecture, ITCA reduces the off estimation down to 2.4 % (2 cores), 3.7 % (4 cores) and 2.8 % (8 cores), while the CA presents a 7.0 %, 13 % and 16 % off estimation, respectively.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625 and grant BES-2008-003683, by the HiPEAC Network of Excellence (IST-004408) and a Collaboration Agreement between IBM and BSC with funds from IBM Research and IBM Deep Computing organizations. The authors are grateful to Alper Buyuktosunoglu who co-authored the original version of this paper and to Enrique Fernández from the University of Las Palmas of Gran Canaria for his help setting up the Intel Xeon Quad-Core processor.

Referencias

- [1] Standard Performance Evaluation Corporation, "SPEC CPU 2000 benchmark suite," <http://www.spec.org>.
- [2] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," in *IEEE Computer*, vol. 30, no. 9, 1997.
- [3] C. Luque, M. Moreto, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs," in *PACT*, 2009.
- [4] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable Performance in SMT Processors: Synergy between the OS and SMTs," in *IEEE Trans. Computers*, vol. 55, no. 7, 2006.
- [5] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt, "QoS policies and architecture for cache/memory in CMP platforms," in *SIGMETRICS*, 2007.
- [6] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *MICRO*, 2006.
- [7] J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, 1970.
- [8] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero, "The MPsim Simulation Tool," in UPC, Tech. Rep. UPC-DAC-RR-CAP-2009-15, 2009.
- [9] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," in *IEEE PACT*, 2001.
- [10] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Architectural Support for Real-Time Task Scheduling in SMT Systems," in *CASES*, 2005.
- [11] D. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *ISCA*, 1995.
- [12] P. Mackerras, T. S. Mathews, and R. C. Swanberg, "Operating system exploitation of the POWER5 system," in *IBM J. Res. Dev.*, vol. 49, no. 4/5, 2005.
- [13] M. S. Floyd, S. Ghiasi, T. W. Keller, K. Rajamani, F. L. Rawson, J. C. Rubio, and M. S. Ware, "System power management support in the IBM POWER6 microprocessor," in *IBM J. Res. Dev.*, vol. 51, no. 6, 2007.